AD-A104 650 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
PROGRAMMING WITH A RELATIONAL CALCULUS.(U)
SEP 81 B J MACLENNAN
UNCLASSIFIED NPS52-81-013 NL

1 OF 1
AD A
104650

END
DATE
FILMED
10-81
DTIC

② LEVEL II

AD A104650

NPS52-81-013

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
ELECTE
SEP 2 9 1981
S D
B

PROGRAMMING WITH A RELATIONAL CALCULUS

B. J. MacLennan. 80/10/21.

September 81

Approved for public release; distribution unlimited

Prepared for:

Naval Postgraduate School
Monterey, CA 93940

DTIC FILE COPY

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

Jack R. Borsting
Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

[illegible]
BRUCE J. MacLENNAN
Assistant Professor of
Computer Science

Reviewed by:

[illegible]
GORDON H. BRADLEY, Chairman
Department of Computer Science

Released by:

William M. Tolles
WILLIAM M. TOLLES
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-013 — 2. GOVT ACCESSION NO. AD-A104650	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROGRAMMING WITH A RELATIONAL CALCULUS.	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) B. J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N RR000-01-10 N0001481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE 3 September 1981
	13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Relational Programming, Functional Programming, Relational Algebra, Relational Calculus, Relations, Applicative Languages, Combinators, Very-High-Level Languages, Logic Programming.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This report describes the concept of programming in a relational calculus. This is a style of programming in which entire relations are manipulated rather than individual data, and in which the program itself is represented as a relation. Thus relational programming is more general than functional programming in three respects. First, it is more general because relations subsume functions. Second, it is more general because the same objects, viz. relations are used to represent both the program and the data. Finally, since complex data structures are easily represented as relations, relational programming can

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

14 16 17 1 7 10 17 11 16 RR000-01 251450

manipulate with facility a much wider class of structures that other very-high-level languages.

PROGRAMMING WITH A RELATIONAL CALCULUS

B. J. MacLennan. 80/10/21.
Naval Postgraduate School
Monterey, CA 93940

CONTENTS

1. Introduction

In this report* we discuss relational programming, i.e. a style of programming in which entire relations are manipulated rather than individual data. This is analogous to functional programming [1], wherein entire functions are the values manipulated by the operators. We will see that relational programming subsumes functional programming because every function is also a relation. It is appropriate at this point to discuss why we have chosen to investigate relational programming. The reader can find a shorter introduction to relational programming in [12].

As we have noted, relational programming subsumes functional programming; hence, anything that can be done with functional programming can be done with relational programming. Furthermore, relational programming has many of the advantages of functional programming: for instance, the ability to derive and manipulate programs by algebraic manipulation. A well developed algebra of relations dates back to Boole's original work and has been extensively studied since then. Although relations are more general than functions, their laws are often simpler. For instance, $(fg)^{-1} = g^{-1}f^{-1}$ is true for all relations, but true only for functions that are one-to-one. Also, relational programming more directly supports non-linear data structures, such as trees and graphs, than does functional programming. In relational programming the basic data values are themselves relations, whereas in functional programming there is a separate class of objects (lists) used for data structures. One final reason for investigating relational programming is that it provides a possible paradigm for utilizing associative and active memories. As a teaser for what is to come, we present the following example of a relational program. We will take a text T, represented as an array of words (i.e., T:i is the i-th word), and generate a frequency table F so that F:w is the number of occurences of word w in T. Now we will see (section 4) that $\overleftarrow{T}$:w is the set of all indices of the word w. If we let #:C be the cardinality of a class, then the number of indices (occurences) of w is just #:($\overleftarrow{T}$:w). Therefore we can write F = #$\overleftarrow{T}$ (Section 7).

2. Classes and Relations

2.1 basic concepts

Our relational calculus will deal with three sorts of things: individuals, classes and relations. These can best be illustrated by example. If 'x' is the name of an individual and 'C' is the name of a class, then 'x∈C' means that the individual denoted by 'x' is a member of the class denoted by 'C' (i.e.,

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

that x has property C). Thus 'Aristotleϵman' would indicate that Aristotle is a man, and '2ϵeven' would mean that 2 is an even number. (The symbol 'ϵ' is an abbreviation for ἐστι, which is the Greek word for 'is'.)

If 'x' and 'y' are names of individuals and 'R' is the name of a relation, then 'x R y' means that x bears the relation R to y. For example,

Aristotle student Plato

means that Aristotle is a student of Plato. Also, '2 < 3' means that 2 bears the less-than relation to 3, i.e., that 2<3. Where there is little chance of confusion, 'x R y' will be written 'xRy' and 'xϵP' will be written 'xP'. The notation that we have introduced above will be extended to classes of classes, classes of relations, relations among classes, relations among relations, etc.

2.2 relational descriptions

If 'S(x)' is a sentence involving 'x', then a class description is an expression of the form '$\hat{x}(S(x))$'. This denotes the class of all individuals, a, for which S(a) is true, i.e.,

$$a \in \hat{x}(S(x)) \leftrightarrow S(a)$$

Similarly, if 'S(x,y)' is a sentence involving 'x' and 'y', then '$\hat{x}\hat{y}(S(x,y))$' is a relation description which holds between a and b whenever S(a,b) is true, i.e.,

$$a\ [\ \hat{x}\hat{y}(S(x,y))\ ]\ b \leftrightarrow S(a,b)$$

To illustrate this notation we will define the converse of a relation.

2.3 converse

The relation R^{-1} is called the converse of R, i.e. $xR^{-1}y \leftrightarrow yRx$. Using our notation for descriptions we can define,

$$R^{-1} = \hat{x}\hat{y}(yRx)$$

As an example of a relation among relations, we define "'" as the relation that holds between converses:

$$r\ '\ s \leftrightarrow r=s^{-1}$$

Hence,

$$' = \hat{r}\hat{s}(r=s^{-1})$$

Some examples of converses are:

$$\text{parent}^{-1} = \text{child}$$
$$\leq^{-1} = \geq$$

The following are easily proved properties of the converse:

$(r^{-1})^{-1} = r$
r's $\leftrightarrow$ s'r
$'^{-1} = '$

2.4 arrow diagrams

Relations can be portrayed by "arrow diagrams" (Haase diagrams). In such a diagram there is a node for each individual related by the relation and an arrow from x to y whenever xRy. For instance,

$$R = \begin{tikzcd} & & c \arrow[dl] & & \\ a & b \arrow[l] \arrow[rrr, bend right] & & d \arrow[ll] & e \arrow[l] \arrow[loop right] \end{tikzcd}$$

represents the relation R such that

bRa, cRb, dRb, eRd, eRe, bRe

and ~xRy for all other cases. The effect of the converse operator is to reverse all of the arrows. Hence, R^{-1} is diagrammed:

$$R^{-1} = \begin{tikzcd} & & c & & \\ a \arrow[r] & b \arrow[ur] \arrow[rr] & & d \arrow[r] & e \arrow[lll, bend left] \arrow[loop right] \end{tikzcd}$$

2.5 tables

Relations can often be viewed as tables. For instance, the relation R of the previous section can be shown as a table:

R	
b	a
c	b
d	b
e	d
e	e
b	e

Of course, it makes no difference in what order we write the rows of the table.

The converse of a relation is obtained by simply exchanging the columns of the table:

R^{-1}

a	b
b	c
b	d
d	e
e	e
e	b

Of course, classes are represented by one column tables. For instance the class C of primes less than ten is:

C

2
3
5
7

3. Domains

We often need to talk of the individuals that can occur on the right or left of a relation. We say that x is a left-member of R whenever there is a y such that xRy.

$$x \text{ Lm } R \leftrightarrow \exists y(xRy)$$

For instance, if 'x parent y' means that x is a parent of y, then 'Socrates Lm parent' means that Socrates is a parent. Right-member and member are defined analogously:

$$y \text{ Rm } R \leftrightarrow \exists x(xRy)$$
$$z \text{ Mm } R \leftrightarrow z \text{ Lm } R \vee z \text{ Rm } R$$

These satisfy the identities:

$$x \text{ Lm } R \leftrightarrow x \text{ Rm } R^{-1}$$
$$y \text{ Rm } R \leftrightarrow y \text{ Lm } R^{-1}$$

4. Functions

4.1 basic concepts

Functions and relations are closely related. Consider the predecessor relation, 'pred':

$$x \text{ pred } y \leftrightarrow x = y-1$$

Thus, x pred y says that x is the predecessor of y. The corresponding arrow diagram is:

$$\begin{tikzcd} 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r, dashed] & \cdots \end{tikzcd}$$

and the corresponding table is:

1	2
2	3
3	4
4	5
:	:
:	:

since 1 pred 2, 2 pred 3, etc. Notice that, in this case, for each right member x there is a unique left member y such that y pred x. This y can be written using Whitehead and Russell's [16] definite description:

$$\iota y(y \text{ pred } x)$$

This can be read: the y such that y is a predecessor of x. A more convenient way to write this is:

$$\text{pred}:x$$

In general, R:x means "the unique y such that y R x, i.e.

$$R:x \quad = \quad \iota y(yRx)$$

This notation is meaningful only if there is a unique y such that yRx, i.e.

$$yRx \wedge zRx \Rightarrow y=z$$

That is, there is only one arrow leading to x. When this condition is satisfied for all x we call R left univalent, symbolized by 'lun':

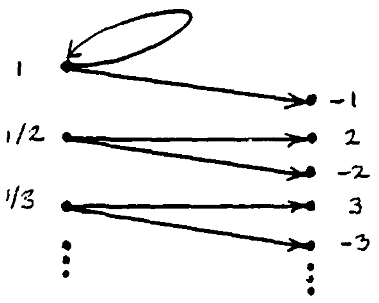
$$R\in\text{lun} \leftrightarrow \forall xyz[\ yRx \wedge zRx \Rightarrow y=z\ ]$$

The left univalent relations are more commonly called functions. In a left univalent relation there is exactly one arrow leading to each node. Consider the "absolute reciprocal" relation:

$$xRy \leftrightarrow x = |1/y|$$

This is diagrammed:

1	1
1	-1
1/2	2
1/2	-2
1/3	3
1/3	-3
:	:
:	:



Since R∈lun it is meaningful to write R:x, so we observe R:(-3) = 1/3. We can find R:x by following back the arrow pointing to x or by looking down the right column for x and taking the corresponding element from the left column.

The concepts of right univalence and bi-univalence are defined analogously:

$$R \in run \leftrightarrow \forall xyz[\ xRy \land xRz \Rightarrow y=z\ ]$$
$$R \in bun \leftrightarrow R \in lun \land R \in run$$

Bi-univalent relations are also called bijections and one-one mappings.

4.2 higher level functions

Of course, the converse of a function is not necessarily a function. The 'sin' relation, defined so that y sin x means that y is the sine of x, is left univalent but not right univalent. Hence, we can write either y=sin:x or y sin x, but can express the arcsine only by:

$$x \sin^{-1} y$$

The notation $\sin^{-1}$:y is meaningless. Since f:x is meaningful only when f∈lun we will be careful to write f:x only when we have previously shown (or it is obvious) that f∈lun and x Rm f.

The fact that f:x may be meaningless makes it convenient to use several other relations derived from f. One of these is the <u>plural description</u>. If F is any relation and C is a class then <u>F!:C</u> is the set of all y such that yFx for some x in C, i.e.,

$$F! = \hat{z}\hat{C}\{z = \hat{y}[\exists x(yFx \land x \in C)]\}$$

The tabular interpretation of F!:C is simple:

F	
:	
y1	x1
:	
yn	xn
:	

C
x1
x2
:
xn

F!:C
y1
y2
:
yn

We see that, if F is any function, then F!:S is the image of the class S under that function. Notice that the operation F!:S is defined for all relations F and classes S, regardless of whether F∈lun or the members of S are right members of F. For these reasons, it is generally safer to write F!:C than F:x.

Related ideas are the image and converse image of an individual. If R is a relation, then c $\overrightarrow{R}$ x means that c is the class of individuals related to x. This class is called the <u>referents</u> of x, and is defined:

$$\overrightarrow{R}:x \quad = \quad \hat{y}(yRx)$$

The converse idea is that of the <u>relata</u> of y:

$$\overleftarrow{R}:y \quad = \quad \hat{x}(yRx)$$

Like the plural description, $\overrightarrow{R}$ and $\overleftarrow{R}$ are defined for all R and all arguments.

Consider now the function $\overrightarrow{\equiv}$:

$$\overrightarrow{\equiv}:x \quad = \quad \hat{y}(y=x)$$

Hence, $\overrightarrow{\equiv}$:y is just the <u>unit class</u> containing y. Russell and Whitehead [16] write this ι:y. Conversely, if C is a single element class, then $(\overrightarrow{\equiv}^{-1})$:C selects the unique member of that class:

$$(\overrightarrow{\equiv}^{-1}):C \quad = \quad \iota x(x \in C)$$

It is thus a uniqueness filter. We will write this as Θ:C:

$$\Theta \quad = \quad \overrightarrow{\equiv}^{-1}$$

The expression Θ:C can be read "the C".

We will occasionally need to refer to the relations that hold between R and $\overrightarrow{R}$ or R and $\overleftarrow{R}$, which we write $\overrightarrow{-}$ and $\overleftarrow{-}$, respectively:

$$R \overrightarrow{-} S \quad \longleftrightarrow \quad R = \overrightarrow{S}$$
$$R \overleftarrow{-} S \quad \longleftrightarrow \quad R = \overleftarrow{S}$$

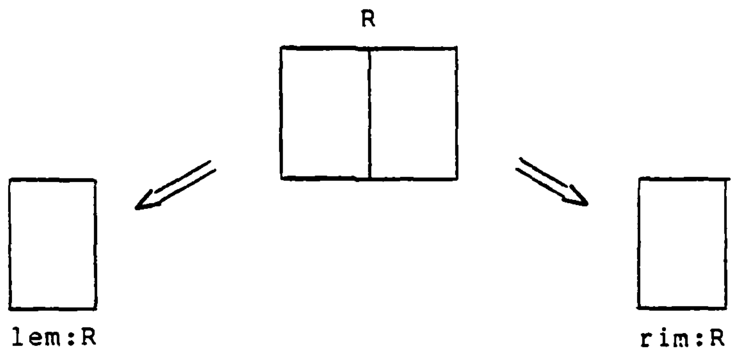
The following are some properties of these operations:

$$\overrightarrow{R^{-1}} = \overleftarrow{R}$$
$$\overleftarrow{R^{-1}} = \overrightarrow{R}$$
$$\overrightarrow{R}:y = R!:(\overrightarrow{\equiv}:y)$$
$$\overrightarrow{R} = \overrightarrow{-}:R$$
$$\overleftarrow{R} = \overleftarrow{-}:R$$

It is often convenient to have names for domain extracting functions, e.g., lem:R is the class of left members of R. These are simply defined using images:

$$\text{lem} = \overrightarrow{\text{Lm}}$$
$$\text{rim} = \overrightarrow{\text{Rm}}$$
$$\text{mem} = \overrightarrow{\text{Mm}}$$

Of course the right and left members of a relation can be obtained by taking its right and left columns, respectively, and deleting duplicates.



5. Boolean Operations

5.1 logical connectives

We will next investigate ways of combining relations and classes. The simplest methods are just abstractions of the logical connectives used between propositions: Therefore, we define the intersection, union, negation and difference of classes and relations:

$$x(S \wedge T) \leftrightarrow xS \wedge xT$$
$$x(R \wedge S)y \leftrightarrow xRy \wedge xSy$$

$$x(S \vee T) \leftrightarrow xS \vee xT$$
$$x(R \vee S)y \leftrightarrow xRy \vee xSy$$

$$x(-S) \leftrightarrow -(xS)$$
$$x(-R)y \leftrightarrow -(xRy)$$

$$x(S-T) \leftrightarrow xS \wedge -(xT)$$
$$x(R-S)y \leftrightarrow xRy \wedge -(xSy)$$

$$x(S \rightarrow T) \leftrightarrow xS \rightarrow xT$$
$$x(R \rightarrow S)y \leftrightarrow xRy \rightarrow xSy$$

As an example of the use of these operations, consider our previous definition of Mm:

$$z \text{ Mm } R \leftrightarrow z \text{ Lm } R \vee z \text{ Rm } R$$

Using the union operation this can be written:

$$Mm = Lm \vee Rm$$

Similarly,

$$bun = lun \wedge run$$

The logical connectives satisfy the usual properties of a Boolean algebra (e.g., DeMorgan's theorem).

As an example of the use of these operations, we will define the closed interval function, m..n, which is the set of integers m, m+1, ..., n. It is just:

$$m..n = \overrightarrow{\geq}:m \wedge \overrightarrow{\leq}:n$$

where $\geq$ and $\leq$ are the relations on integers.

5.2 empty classes and relations

It is useful to have names for the empty class and relation:

$$\Phi = \hat{x}(x \neq x)$$
$$\ominus = \hat{x}\hat{y}(x \neq x)$$

Hence, $x\Phi$ is always false, as is $x\ominus y$. These are most often used for stating properties of relations and classes. For instance,

$$S \wedge T = \Phi$$

means that classes S and T have no members in common.

The universal classes and relations are also useful:

$$\bar{\Phi} = -\Phi$$
$$\bar{\ominus} = -\ominus$$

For instance,

$$S \vee T = \bar{\Phi}$$

means that every individual is either a member of S or of T. Notice that the class of the right members of a relation is just the image of the universe under that relation, i.e.,

$$\begin{aligned} rim:R &= R!:\bar{\Phi} \\ lem:R &= (R^{-1})!:\bar{\Phi} \\ mem:R &= (R \vee R^{-1})!:\bar{\Phi} \end{aligned}$$

5.3 Cartesian product

It is often useful to have the maximum relation that can hold between two classes, i.e., the Cartesian product of those classes. This is defined:

$$S \ast T = \hat{x}\hat{y}(xS \wedge yT)$$

The Cartesian product satisfies the following properties:

$$\begin{aligned}
(s \ast t)^{-1} &= t \ast s \\
\text{lem}:(s \ast t) &= s \\
\text{rim}:(s \ast t) &= t \\
\text{mem}:(s \ast t) &= s \vee t
\end{aligned}$$

$$\begin{aligned}
s \ast (t \wedge u) &= (s \ast t) \wedge (s \ast u) \\
s \ast (t \vee u) &= (s \ast t) \vee (s \ast u) \\
s \ast (t - u) &= (s \ast t) \wedge (s \ast -u) \\
s \ast (t \rightarrow u) &= (s \ast -t) \vee (s \ast u)
\end{aligned}$$

$$\begin{aligned}
\bar{\Theta} &= \bar{\Phi} \ast \bar{\Phi} \\
s \ast \Phi &= \Phi \ast s = \Phi \ast \Phi = \Theta \\
s \ast t &= (s \ast \bar{\Phi}) \wedge (\bar{\Phi} \ast t)
\end{aligned}$$

5.4 subset relation

Finally, we define the subclass and subrelation operations:

$$\begin{aligned}
S\underline{C}T &\leftrightarrow \forall x(xS \rightarrow xT) \\
R\underline{C}S &\leftrightarrow \forall xy(xRy \rightarrow xSy)
\end{aligned}$$

The following are true:

$$\begin{aligned}
&s\underline{C}t \rightarrow (s \ast u)\underline{C}(t \ast u) \\
&s\bar{\underline{C}}t \rightarrow (r \ast s)\bar{\underline{C}}(r \ast t) \\
&s\bar{\underline{C}}t \wedge u\underline{C}v \rightarrow (s \ast u)\underline{C}(t \ast v) \\
&r\bar{\in}\text{Rel} \rightarrow r\underline{C}\bar{\Theta} \\
&r\in\text{Rel} \rightarrow \Theta\bar{\underline{C}}r \\
&s\in\text{Cls} \rightarrow s\bar{\underline{C}}\bar{\Phi} \\
&s\in\text{Cls} \rightarrow \Phi\bar{\underline{C}}s
\end{aligned}$$

where Cls is the class of all classes and Rel is the class of all relations (we are ignoring typing here). These can be defined:

$$\begin{aligned}
\text{Rel} &= \overrightarrow{\underline{C}}:\bar{\Theta} = \overleftarrow{\underline{C}}:\Theta \\
\text{Cls} &= \underline{C}:\bar{\Phi} = \underline{C}:\Phi
\end{aligned}$$

6. Limiting and Restriction

It is often useful to limit the left or right domain of a relation. Consider the relation $x \sin^{-1} y$, which means that x is an arcsine of y. We cannot write $x = \sin^{-1}:y$ because $\sin^{-1}$ is not left univalent (i.e. it is not a function). If we restrict y, the argument of sin, to the range $-\pi/4$ to $\pi/4$, then there is a unique x such that x $\sin^{-1}$ y. Let S be the class of reals in the range $-\pi/4$ to $\pi/4$:

$$xS \longleftrightarrow (-\pi/4 \leq x) \wedge (x \leq \pi/4)$$

then we will write

$$\sin \rangle S$$

for the sine function with its arguments restricted to S. This function is bi-univalent, so it is invertible. If we call the inverse of this restricted sine Arcsin:

$$\text{Arcsin} = (\sin \rangle S)^{-1}$$

then it is perfectly meaningful to write Arcsin:x (if x Lm sin). The right-restriction operation is defined:

$$x(R \rangle S)y \longleftrightarrow xRy \wedge yS$$

The left-restriction is defined analogously:

$$x(S \langle R)y \longleftrightarrow xS \wedge xRy$$

These notations can be combined to restrict both domains:

$$x\ (S \langle R \rangle T)\ y \longleftrightarrow xS \wedge xRy \wedge yT$$

The combination $s \langle R \rangle s$ is so common that a special notation is provided for it:

$$R \Join s = s \langle R \rangle s$$

For instance, $< \Join P$, where $xP \longleftrightarrow x>0$, is the less-than relation restricted to positive numbers.

The restriction operations are easily defined in terms of intersection and Cartesian product:

$$\begin{aligned} s \langle r \rangle t &= r \wedge (s \times t) \\ r \Join s &= r \wedge (s \times s) \\ s \langle r &= r \wedge (s \times \bar{\Phi}) \\ r \rangle s &= r \wedge (\bar{\Phi} \times s) \end{aligned}$$

C
x1
x2
:
xn

R	
:	:
x1	y1
:	:
x2	y2
:	:
xn	yn
:	:

$C \triangleleft R$	
x1	y1
x2	y2
:	:
xn	yn

Other properties satisfied by these operations are:

$$\begin{aligned}
s \bowtie t &= s \triangleleft \bar{\theta} \triangleright t \\
\mathrm{lem}:(s \triangleleft r) &= s \wedge \mathrm{lem}:r \\
\mathrm{rim}:(r \triangleright s) &= s \wedge \mathrm{rim}:r \\
\mathrm{lem}:(r \triangleright s) &= r!:s \\
\mathrm{rim}:(s \triangleleft r) &= r^{-1}!:s
\end{aligned}$$

$$\begin{aligned}
(s \triangleleft r)^{-1} &= (r^{-1}) \triangleright s \\
(s \triangleleft r \triangleright t)^{-1} &= t \triangleleft (r^{-1}) \triangleright s \\
(r \times s)^{-1} &= (r^{-1}) \times s
\end{aligned}$$

$$\begin{aligned}
\overrightarrow{r \triangleright s} &= \vec{r} \triangleright s \\
\overleftarrow{s \triangleleft r} &= s \triangleleft \overleftarrow{r}
\end{aligned}$$

$$\begin{aligned}
r \triangleright s \wedge r \triangleright t &= r \triangleright (s \wedge t) \\
r \triangleright s \vee r \triangleright t &= r \triangleright (s \vee t) \\
(r \bowtie \bar{\Phi}) \triangleright s &= r \bowtie s
\end{aligned}$$

7. Relative Product

If xRy is the relation "x is a son of y" and xSy is the relation "x is a brother of y", then the relative product, R|S, is the relation "x is a son of a brother of y." More formally,

$$R|S = \hat{x}\hat{z}\{\exists y(xRy \wedge ySz)\}$$

Where there is little chance of confusion, we will write RS for R|S. If f and g are functions it is easy to see that f|g is the composition of these functions:

$$\begin{aligned}
& x = fg:z \\
\leftrightarrow\ & x\ fg\ z \\
\leftrightarrow\ & \exists y(x\ f\ y \wedge y\ g\ z) \\
\leftrightarrow\ & \exists y(x=f:y \wedge y=g:z) \\
\leftrightarrow\ & x = f:(g:z)
\end{aligned}$$

Hence, $fg:x = f:(g:x)$.

It is convenient to have a notation for relative products of a relation with itself. For instance, the "grandparent" relation can be written parent|parent, which we abbreviate parent^2. In general,

$$R^0 = = \hat{x}(\text{mem}:R)$$
$$R^1 = R$$
$$R^{n+1} = (R^n)R = R(R^n)$$
$$R^{-n} = (R^n)^{-1}$$

Some obvious properties of the relative product are:

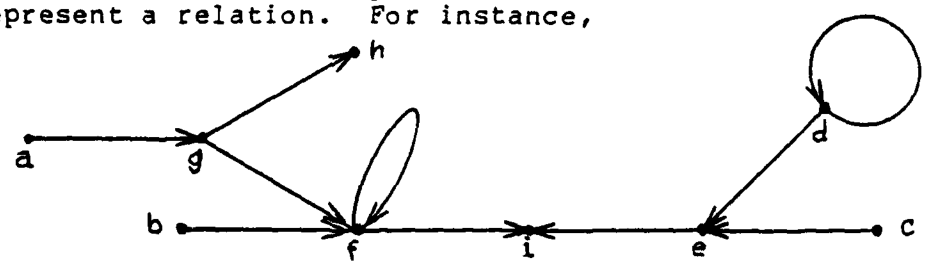
$$(rs)t = r(st)$$
$$r(s \vee t) = rs \vee rt$$
$$(r \vee s)t = rt \vee st$$
$$r(s \wedge t) \subseteq rs \wedge rt$$
$$(r \wedge s)t \subseteq rt \wedge st$$
$$\exists(rs) \leftrightarrow \exists(\text{rim}:r \wedge \text{lem}:s)$$

$$(r^{-1})^{-1} = r$$
$$(rs)^{-1} = (s^{-1})(r^{-1})$$
$$r^m r^n = r^{m+n} \quad (m,n>0)$$
$$(r^m)^n = r^{mn} \quad (m,n\geq 0, \text{ or } r\in\text{bun})$$
$$r^m r^n \subseteq r^{m+n} \quad (r\in\overline{\text{bun}})$$
$$rr^{-1} \overline{=} r^{-1}r = r^0 \quad (r\in\text{bun})$$

$$\text{lem}:rs \subseteq \text{lem}:r$$
$$\text{rim}:rs \subseteq \text{rim}:s$$
$$Lm = R\overline{m}'$$
$$Rm = Lm'$$
$$r\theta = \theta r = \theta$$
$$rI = Ir = r \quad \text{where } I = \hat{x}\hat{y}(x=y)$$

8. Structures

We have previously seen the use of arrow diagrams to represent a relation. For instance,



represents the relation R:

R

a	g
b	f
c	e
d	d
d	e
e	i
f	f
f	i
g	f
g	h

8.1 initial and terminal members

Now, notice that the left and right members of R are:

lem:R = { a, b, c, d, e, f, g }
rim:R = { g, f, e, d, i, h }

We define the initial members of R to be those members which are not pointed at by an arrow. Therefore, the initial members of R are the left members that are not right members.

$$\text{init:R} = \overrightarrow{(\text{Lm}-\text{Rm})}\text{:R} = \{a, b, c\}$$

The terminal members of a relation are defined analogously:

$$\text{term:R} = \overrightarrow{(\text{Rm}-\text{Lm})}\text{:R} = \{h, i\}$$

When a relation is used to represent a data structure, the above functions become important.

For instance, a sequence is represented by a relation with the structure:

$$S = \begin{tikzcd} a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r, dash] & \cdots \arrow[r] & a_{n-1} \arrow[r] & a_n \end{tikzcd}$$

In this case init:S is the unit class containing the head (first element) of the relation (i.e., a_1) and term:S is the unit class containing the last element of the sequence (i.e., a_n). Similarly, S≯(-init:S) is the sequence with its first element deleted:

$$\begin{tikzcd} a_2 \arrow[r] & a_3 \arrow[r, dash] & \cdots \arrow[r] & a_{n-1} \arrow[r] & a_n \end{tikzcd}$$

Hence, the following common sequence manipulation functions can be defined (represented by lower and upper case alphas and omegas):

$\alpha{:}S$	=	θ init: S	"first"
$\omega{:}S$	=	θ term: S	"last"
$\Omega{:}S$	=	$S \rangle (\text{-init}{:}S)$	"final"
$A{:}S$	=	$(\text{-term}{:}S) \langle S$	"initial"

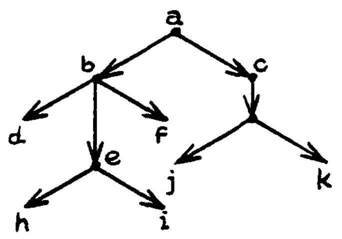
The following properties of these relations are easy to show:

$$\alpha = \omega'$$
$$\omega = \alpha'$$
$$A' = {}'\Omega$$
$$\Omega' = {}'A$$

More operations on sequences are discussed in the next section.

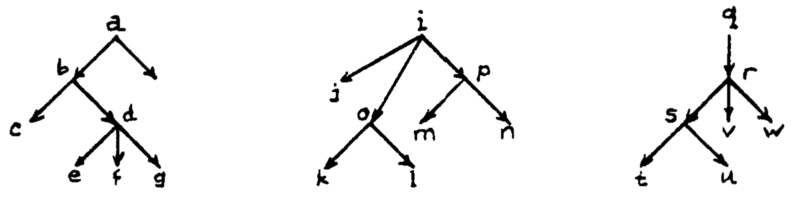
As another example of the use of 'init' and 'term', consider a relation representing a tree:

T =



Then, θ init: T is 'a', the root of the tree, and term:T is {d, h, i, f, j, k}, the leaves of the tree. The result is analogous for forests. Given

F =



the set of roots is init:F and the set of leaves is term:F:

init:F = {a,i,g}
term:F = {c,e,f,g,h,j,k,l,m,n,t,u,v,w}

8.2 <u>higher level operations</u>

The set of nodes whose parent is n is just $\overrightarrow{F^{-1}}$:n. For instance, the set of nodes directly descended from a root is

$$F^{-1}!{:}(\text{init}{:}F) = \{b,h,j,o,p,r\}$$

The set of nodes that point to leaves is

$$F!{:}(\text{term}{:}F) = \{b,d,a,i,o,p,s,r\}$$

These operations can be used for obtaining the maximum and minimum of sets. Suppose '<' is the less-than relation on integers and S is some set of integers, say {3,5,9}. Then

<⨳S =

$$\begin{tikzcd} 3 \arrow[r] \arrow[rr, bend left] & 5 \arrow[r] & 9 \end{tikzcd}$$

Now note that

init:(<⨳S) = {3}
term:(<⨳S) = {9}

Hence, if S is any set of numbers, then the minimum and maximum of this set are:

min:S = α:(<⨳S)
max:S = ω:(<⨳S)

These operations are only defined if S has two or more elements, since an irreflexive relation cannot relate less than two elements. That is, an irreflexive relation when restricted to a unit or empty class becomes the empty relation. Notice that we can select the maximum and minimum based on any relation that is a <u>series</u> (i.e., transitive, irreflexive and connected). If R is any series then α:(R⨳S) is the minimum (relative to R) and ω:(R⨳S) is the maximum.

The following are simple properties of these operations:

init:r = term:(r^{-1})
term:r = init:(r^{-1})
init ⊆ lem
term ⊆ rim
init:(r⨳s) = term:(r^{-1}⨳s)

init:(r V s) ⊆ init:r V init:s
init:r ∧ init:s ⊆ init:(r ∧ s)
term:(r V s) ⊆ term:r V term:s
term:r ∧ term:s ⊆ term:(r ∧ s)
init:(s✱t) = s−t
term:(s✱t) = init:$(s✱t)^{-1}$ = init:(t✱s) = t−s

9. <u>Sequences</u>

9.1 <u>ordinal couples</u>

In this section we will continue the discussion of sequences begun in the last section. We saw that it was easy to define the following operations on sequences:

$$\begin{aligned} \alpha{:}S &= \theta\ \mathrm{init}{:}S \\ \omega{:}S &= \theta\ \mathrm{term}{:}S \\ \Omega{:}S &= (-\mathrm{init}{:}S) \not\prec S \\ A{:}S &= S \not\succ (-\mathrm{term}{:}S) \end{aligned}$$

This provides us with functions for taking sequences apart. We will define the ordinal couple or pair, which puts them together. If x and y are two objects, then 'x,y' is the relation that relates x and y but no other objects.

$$(x,y) \quad = \quad \begin{tikzcd} x \arrow[r] & y \end{tikzcd}$$

That is, $u(x,y)v$ if and only if $u=x$ and $y=v$. This is formally defined by:

$$x,y \quad = \quad \hat{u}\hat{v}(u=x \wedge v=y)$$

This notation will be taken to be right associative, i.e.,

$$x,y,z \quad = \quad x,(y,z)$$

Observe that

$$\begin{aligned} \alpha{:}(x,y) &= x \\ \omega{:}(x,y) &= y \end{aligned}$$

It will occasionally be convenient to write ordinal couples in a vertical format:

$$\begin{pmatrix} x \\ y \end{pmatrix} = (x,y)$$

This notation is extended for relations of more than one pair:

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \vee \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \vee \cdots \vee \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

The class of all the ordinal couples (or pairs) that can be made from the classes S and T is:

$$S \chi T \quad = \quad \hat{p}(\exists xy[x \in S \wedge y \in T \wedge p=(x,y)\ ])$$

There is obviously a close relation between $s \chi t$ and $s * t$. Later we will say that $s * t$ is a Currying of $s \chi t$. Note that

$$(x,y) \in S \chi T \quad \leftrightarrow \quad x\ [S*T]\ y$$

We will define a convenient notation for sequences of two or more elements:

$$\langle\ x_1,\ x_2, \ldots,\ x_n \rangle \quad = \quad (x_1,x_2) \vee (x_2,x_3) \vee \cdots \vee (x_{n-1},x_n)$$

Therefore the sequence <a,b,c,d,e> is just

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[r] & d \arrow[r] & e \end{tikzcd}$$

Also, note that,

$$\langle x_1, x_2, \ldots, x_n\rangle = \begin{pmatrix} x_1 & x_2 & \cdots & x_{n-1} \\ x_2 & x_3 & \cdots & x_n \end{pmatrix} = \begin{array}{|c|c|}\hline x_1 & x_2 \\ \hline x_2 & x_3 \\ \hline \vdots & \vdots \\ \hline x_{n-1} & x_n \\ \hline \end{array}$$

9.2 catenation and consing

If s and t are sequences then we can define an operation 's^t', which is the catenation of s and t. To form this catenation we must hook the last element of s to the first element of t:

$$\begin{tikzcd} s_1 \arrow[r, "\cdots"] & s_m & {}^\wedge & t_1 \arrow[r, "\cdots"] & t_n & = & s_1 \arrow[r, "\cdots"] & s_m \arrow[r] & t_1 \arrow[r, "\cdots"] & t_n \end{tikzcd}$$

Therefore x [s^t] y if and only if x s y, or x t y, or $x=\omega$:s and $y=\alpha$:t. Hence,

$$s\hat{\ }t = s \vee (\omega{:}s, \alpha{:}t) \vee t$$

The catenation operation is only defined for sequences, which are required to have at least two elements (since an irreflexive relation with less than two elements is the empty relation). Note that we can extend the definition of sequences so as to allow length one sequences by making the relation reflexive.

$$s \vee (= x\ \mathrm{mem}{:}s)$$

$$\begin{tikzcd} s_1 \arrow[loop above] \arrow[r] & s_2 \arrow[loop above] \arrow[r] & s_3 \arrow[loop above] \arrow[r, "\cdots"] & s_n \arrow[loop above] \end{tikzcd}$$

The one element sequence is then:

$$\begin{tikzcd} s_0 \arrow[loop above] \end{tikzcd} = (s_0, s_0)$$

The full ramifications of this definition of sequence have not been investigated.

How do we add a single element to the left or right of a sequence? The "cons left" and "cons right" operations are easy to define:

$$\begin{tikzcd} x & \mathrm{cl} & s_1 \arrow[r, "\cdots"] & s_n & \Rightarrow & x \arrow[r] & s_1 \arrow[r, "\cdots"] & s_n \end{tikzcd}$$

$$x \text{ cl } s = (x, \alpha\!:\!s) \vee s$$
$$s \text{ cr } y = s \vee (\omega\!:\!s,\ y)$$

It is easy to show that if s is a sequence, then:

$$\alpha\!:\!(x \text{ cl } s) = x$$
$$\Omega\!:\!(x \text{ cl } s) = s$$

$$\omega\!:\!(s \text{ cr } y) = y$$
$$A\!:\!(s \text{ cr } y) = s$$

$$(\alpha\!:\!s) \text{ cl } (\Omega\!:\!s) = s, \quad \text{if } \#\!:\!s > 2$$
$$(A\!:\!s) \text{ cr } (\omega\!:\!s) = s, \quad \text{if } \#\!:\!s > 2$$

Also, if s is a sequence, then s V (ω:s, α:s) is a ring formed by joining the last element of s to the first element.

If s is a sequence, then s^{-1} is the reverse of s. Hence,

$$\text{rev}\!:\!s = s^{-1}$$

$$\alpha\!:\!s = \omega\!:\!s^{-1}$$
$$\omega\!:\!s = \alpha\!:\!s^{-1}$$

$$A\!:\!s = (\Omega\!:\!s^{-1})^{-1}$$
$$\Omega\!:\!s = (A\!:\!s^{-1})^{-1}$$

$$(s^\wedge t)^{-1} = t^{-1}{}^\wedge s^{-1}$$
$$(x \text{ cl } s)^{-1} = s^{-1} \text{ cr } x$$
$$(s \text{ cr } x)^{-1} = x \text{ cl } s^{-1}$$

$$(x,y)^{-1} = (y,x)$$
$$\langle x_1, x_2, \ldots, x_n\rangle^{-1} = \langle x_n, \ldots, x_2, x_1\rangle$$
$$\begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix}^{-1} = \begin{pmatrix} y_1 & y_2 & \cdots & y_n \\ x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

If S is a sequence and x Mm S, then S^{-1}:x is the successor of x in S and S:x is the predecessor of x in S (if these exist).

$$S^{-1}\!:\!x = \text{successor of } x \text{ in } S$$
$$S\!:\!x = \text{predecessor of } x \text{ in } S$$

These are convenient ways of moving around within a sequence. Also, note that if s is a subsequence of t then $s \subseteq t$.

Some additional identities are:

$$\binom{x}{y}\binom{y}{z} = \binom{x}{z}$$

$$\alpha!\!:\!(S \times T) = S$$
$$\omega!\!:\!(S \times T) = T$$

Finally, we will state the formal definition of a sequence: a relation is a sequence if it is a connected irreflexive bijection. That is,

$$\text{sequence} = \text{connex} \wedge \text{irrefl} \wedge \text{bun}$$

$$s \in \text{irrefl} \leftrightarrow s^0 \subseteq s^{-1}$$
$$s \in \text{connex} \leftrightarrow \text{lem:s} = \text{init:s} \vee \text{rim:s} \wedge \text{rim:s} = \text{term:s} \vee \text{lem:s}$$

10. Binary Operations

10.1 basic concepts

In this section we will discuss our approach to binary operations - that is, to functions with two arguments and one result. We have already seen how unary functions are connected to relations. For instance, we can write the fact that y is the sine of x by either:

$$y \sin x$$

or

$$y = \sin{:}x$$

Since we only deal with binary relations, we will have to have a new convention for handling binary functions. This convention is: we will combine the two arguments of an operation into a pair. For instance, we can define a relation 'sum' such that

$$x \text{ sum } (y,z)$$

if and only if x is the sum of y and z. More formally:

$$\text{sum} = \hat{x}\hat{a}(a=(y,z) \wedge x=y+z)$$

We can use our colon convention as usual, e.g.,

$$x = \text{sum:}(y,z) \leftrightarrow x \text{ sum } (y,z)$$

Now, it would be inconvenient to have to invent names, such as 'sum', for each operation, such as '+'. Hence, we will adopt a systematic convention for making such names: either placing the conventional infix symbol for the operation in parentheses or underlining the symbol. For instance,

$$x\underline{+}(y,z) \leftrightarrow x = \underline{+}{:}(y,z) \leftrightarrow x = y+z$$

In fact, if π is any infix operation symbol, we will explicitly define its meaning by

$$x\pi y \quad = \quad \underline{\pi}:(x,y)$$

This notation will permit us to manipulate in a more regular fashion the usual arithmetic operations (+, -, *, /) as well as the relational operations (e.g. $\wedge$, V, χ, $\not\subset$, $\not\supset$, $\not\times$, $\not*$, ','). For instance, if S is a class of classes, then

$$(\wedge)!:S\chi S$$

is the class of all pairwise intersections of members of S.

10.2 operations on binary operations

It is often convenient to be able to generate simple relations from a binary operation. Following Russell and Whitehead, let π represent any binary operation. We define:

$$\pi z \quad = \quad \hat{x}\hat{y}(x = y\pi z)$$
$$y\pi \quad = \quad \hat{x}\hat{z}(x = y\pi z)$$

Hence,

$$x(-1)y \quad \leftrightarrow \quad x = y-1$$

therefore (-1) is the predecessor relation. Similarly,

$$x(1+)y \quad \leftrightarrow \quad x = 1+y$$

therefore (1+) (or (+1)) is the successor relation. These can be used as functions:

$$(-1):x \quad = \quad x-1$$
$$(+1):x \quad = \quad x+1$$

This convention makes it very easy to form more complex functions. For instance, if we want

$$f:x \quad = \quad \sin:(1/x)$$

then we can define

$$f \quad = \quad \sin(1/)$$

To see that this works:

$$\begin{aligned} f:x &= [\sin(1/)]:x \\ &= \sin:[\ (1/):x\ ] \\ &= \sin:[\ 1/x\ ] \end{aligned}$$

Now observe the action of the (x,) and (,y) functions:

$$(x,):y \quad = \quad (x,y)$$
$$(,y):x \quad = \quad (x,y)$$

Therefore, for any binary operation π (except ',') we can define

$$\begin{aligned} \pi z &= \underline{\pi}(,z) \\ y\pi &= \underline{\pi}(y,) \end{aligned}$$

Let's see why this works:

$$\begin{aligned} (y\pi):z &= [\underline{\pi}(y,)]:z \\ &= \underline{\pi}:[(y,):z] \\ &= \underline{\pi}:[y,z] \\ &= y\pi z \end{aligned}$$

$(\pi z):y$ is analogous. In general, if f is a binary function, then f(x,) and f(,y) are the "partially instantiated" unary functions. This is the effect of Curry and Feys "B" combinator [5].

Since S^{-1} is the reverse of a sequence, π|' is the reverse form of an operation. For instance, $\underline{-}$' is the reverse subtract operation:

$$\begin{aligned} \underline{-}':(x,y) &= \underline{-}:(':(x,y)) \\ &= \underline{-}:(y,x) \\ &= y-x \end{aligned}$$

Thus $\underline{-}$' can be read "subtract from" and $\underline{/}$' can be read "divide into". This is Curry and Feys "C" combinator (see the next section).

11. Combinators

In this section we will discuss several powerful operations for manipulating relations. These are called combinators because of their similarity to the combinators of Curry and Feys [5].

The first combinator we will discuss is the paralleling of relations, $\frac{R}{S}$, which is defined:

$$\begin{pmatrix} u \\ v \end{pmatrix} \begin{pmatrix} \frac{R}{S} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leftrightarrow uRx \wedge vSy$$

So, if f and g are functions,

$$\begin{pmatrix} \frac{f}{g} \end{pmatrix} : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f:x \\ g:y \end{pmatrix}$$

Hence, $\frac{f}{g}$ is the element-wise combination of f and g. For example, if we want f:(x,y) = sin:x + sin:y, we can write

$$f = (+)\frac{\sin}{\cos}$$

since

$$\begin{aligned} f:(x,y) &= (+)\frac{\sin}{\cos}:\binom{x}{y} \\ &= (+):\left(\frac{\sin}{\cos}:\binom{x}{y}\right) \\ &= (+):\begin{pmatrix}\sin:x \\ \cos:y\end{pmatrix} \\ &= \sin:x + \cos:y \end{aligned}$$

One of the simplest combinators described by Curry and Feys is the <u>elementary cancellator</u>, $\underline{K}$, defined so that $\underline{K}:x$ is a function such that $(\underline{K}:x):y = x$ for all y. That is, $\underline{K}$ generates constant functions. Since $\underline{K}:x$ is a relation that relates x to everything, we can define it:

$$\underline{K} = (\ast\bar{\Phi})i$$

where $i = \Theta^{-1}$ is the unit class generator. To see that this works, note that

$$\underline{K}:x = (\ast\bar{\Phi})i:x = (i:x)\ast\bar{\Phi}$$

and therefore that

$$\begin{aligned} u(\underline{K}:x)v &\leftrightarrow u[(i:x)\ast\bar{\Phi}]v \\ &\leftrightarrow u \in i:x \wedge v \in \bar{\Phi} \leftrightarrow u=x \end{aligned}$$

Therefore, $(\underline{K}:x):v = x$.

Another combinator is the <u>elementary duplicator</u>, W, defined so that

$$(W:f):x = f:(x,x)$$

If we define $\triangle:x = (x,x)$ then it is easy to see that $W:f$ is just $f\triangle$. For instance, $(*)\triangle$ is the squaring function:

$$\begin{aligned} (*)\triangle:n &= (*):(\triangle:n) \\ &= (*):(n,n) = n*n = n^2 \end{aligned}$$

It should be clear that Backus' [f,g] combining form is just our $\frac{f}{g}\triangle$, since

$$\frac{f}{g}\triangle:x = \left(\frac{f}{g}\right):\binom{x}{x} = \begin{pmatrix}f:x \\ g:x\end{pmatrix}$$

Since this combination is so common we will adopt a special notation for it:

$$\frac{f}{g}\Big| = \frac{f}{g}\triangle$$

Hence, $\frac{f}{g}\Big|:x = \begin{pmatrix}f:x \\ g:x\end{pmatrix}$

Some of the properties satisfied by these combinators are:

$$\frac{R}{S}\,\frac{T}{U} = \frac{RT}{SU}$$

$$\left(\frac{R}{S}\right)^{n} = \frac{R^{n}}{S^{n}}$$

$$\left.\frac{R}{S}\right|\,T = \frac{RT}{ST}$$

$$\frac{R}{S}\,\left.\frac{T}{U}\right| = \left.\frac{RT}{SU}\right|$$

$$\Delta R = \left.\frac{R}{R}\right|$$

$$\frac{R}{S}\,' = \frac{S}{R} = {}'\,\frac{R}{S}$$

$$'\,\left.\frac{R}{S}\right| = \left.\frac{S}{R}\right|$$

$$\alpha\,\left.\frac{R}{S}\right| = R \nsucc (\overrightarrow{Rm}:S)$$

$$\omega\,\left.\frac{R}{S}\right| = S \nsucc (\overrightarrow{Rm}:R)$$

$$\frac{R}{S} = \left.\frac{R\alpha}{S\omega}\right|$$

$$cl = \left.\frac{\alpha}{\Omega}\right|^{-1}$$

$$cr = \left.\frac{A}{\omega}\right|^{-1}$$

As an example of these combinators it is easy to show that

$$f = (+)\left.\frac{(*)\Delta}{2*}\right|$$

is the function $f:t = t^2+2t$.

Another combinator is the meta-application operator, ::, which corresponds to Curry and Feys' S combinator:

$$(f::g):x = (f:x):(g:x)$$

For instance, [(!)']::init is the operation that gives the set of descendents of roots of a forest, F, since

$$([(!)']::init):F = (F^{-1}!):(init:F).$$

The <u>formalizing</u> <u>combinator</u>, $\underline{\Phi}$, is defined so that

$$(\underline{\Phi}:(f,a,b)):x = f:(a:x,b:x)$$

It is easy to see that

$$\underline{\Phi}:(f,a,b) = f\frac{a}{b}$$

For instance,

$$f = \underline{\Phi}:((+),\ (*)\Delta,\ 2*)$$

is just the function $f:x = x^2+2x$. This can be written more clearly using the notation of our relational calculus:

$$f = (+)\frac{(*)\Delta}{2*}\Big|$$

Another combinator defined by Curry and Feys is the $\underline{\Psi}$ combinator:

$$[\underline{\Psi}:(f,g)]:(x,y) = f:(g:x,g:y)$$

This is simply defined by

$$\underline{\Psi}:(f,g) = f\frac{g}{g}$$

so that

$$\underline{\Psi} = (|)\frac{I}{(/)}\Delta$$

Therefore, if

$$f = \underline{\Psi}:((+),\ (*)\Delta)$$

then $f:x = x^2+y^2$.

One final operation we wish to define in this section is "Currying". This relates a relation to the correponding class of pairs. If S is a class of pairs, then Curry:S, the Currying of S, is the relation R such that xRy if and only if $(x,y)\in S$. Formally,

$$\text{Curry}:S = \hat{x}\hat{y}[(x,y)\in S]$$

The inverse operation, $\text{Curry}^{-1}:R$, is also useful.

Some properties satisfied by these combinators are:

$$\text{Curry}:(S\chi T) = S\ast T$$
$$(\overline{\underline{K}}:x)f = \overline{\underline{K}}:x$$
$$f\overline{(\underline{K}:x)} = \underline{\overline{K}}:(f:x) = \underline{K}f:x$$
$$\Delta\bar{\Delta} = I$$

12. <u>Ancestral Relations</u>

12.1 <u>definition</u>

Carnap [2] defines the relation of a property p being hereditary with respect to a relation r:

$$p \text{ Her } r \iff \forall xy\{x\in p \wedge x\ r\ y \rightarrow y\in p\}$$
$$\iff r^{-1}!:p \subseteq p$$

This leads to the definition of the <u>ancestral</u> <u>of</u> <u>R</u> <u>of</u> <u>the</u> <u>first</u>

<u>kind</u> as that relation which preserves all the hereditary properties of R. This is also called the <u>transitive</u> <u>closure</u> of R:

$$x\ R^*\ y \quad \leftrightarrow \quad x \text{ Mm } \mathbf{R} \wedge \forall p[p \text{ Her } R \wedge x \in p \rightarrow y \in p]$$

For example, if xPy means that x is a parent of y, then xP^*y means that x is an ancestor (or the same as) y. The <u>ancestral</u> <u>of</u> <u>the</u> <u>second</u> <u>kind</u> is also useful:

$$R^+ = R^* | R$$

Thus, P^+ means "ancestor" in the colloquial sense. The easiest way to visualize the meanings of the ancestrals is by their expansion as infinite unions:

$$R^* = R^0 \vee R^1 \vee R^2 \vee R^3 \vee \ldots$$
$$R^+ = R^1 \vee R^2 \vee R^3 \vee R^4 \vee \ldots$$

Here are some useful properties of the ancestrals:

$$R^+ = R^* - (=) = R^* - R^0$$
$$xR^*y \quad \leftrightarrow \quad \exists n[n \geq 0 \wedge xR^ny]$$

$$R^0 \subseteq R^*$$
$$R^n \subseteq R^*, \text{ for } n \geq 0$$
$$R^n \subseteq R^+, \text{ for } n > 0$$

$$R|R^+ = R^+$$
$$R^+ \subseteq R^*$$
$$R^+ = R|R^*$$
$$R^* = R^0 \vee R^+$$

$$(R^*)^{-1} = (R^{-1})^*$$
$$(R^+)^{-1} = (R^{-1})^+$$

$$(r ⨳ s)^* \subseteq r^* ⨳ s$$

Ancestral relations are always transitive. Notice that $\geq$ and $>$ for integers can be defined:

$$\geq \; = \; (1+)^*$$
$$> \; = \; (1+)^+$$

The ancestral "fills out" all of the paths in a structure. For instance, if

$$R = \begin{tikzcd} a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \end{tikzcd}$$

then

$$\begin{tikzcd} R^* = & a_1 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left=50] & a_2 \arrow[r] \arrow[rr, bend left] & a_3 \arrow[r] & a_4 \end{tikzcd}$$

12.2 applications

Suppose that S is a sequence and we wish to find the first member of S which satisfies some property P. First form the closure S^+, so that for any two members of S^+ we can tell which is first. Next, eliminate from S^+ any members that do not satisfy P: $S^+ \times P$. Then, $\propto:(S^+ \times P)$ is the first member of S satisfying P.

Next we will consider a simple character manipulation example: stripping leading blanks from a string. Note that $x\ (y\ \mathrm{cl})^*\ z$ means that x is a result of consing 0 or more y's on the front of z. Hence,

$$z\ [(y\ \mathrm{cl})^*]^{-1}\ x$$

means that z is the result of stripping one or more y's from the front of x. To get the desired result it is only necessary to restrict the left domain of this function to be sequences that don't begin with a y. Suppose Y is the property of beginning with a y:

$$xY \leftrightarrow y = \propto:x \leftrightarrow y \propto x \leftrightarrow x \in \overleftarrow{\propto}:y$$

Therefore, the function to strip leading y's from a sequence is:

$$(-\overleftarrow{\propto}:y) \nless [(y\ \mathrm{cl})^*]^{-1}$$

Before we leave the topic of ancestral relations, it will be useful to investigate their use as a means of iteration. Suppose that F is a function (i.e., left univalent). Then, since

$$F^* = F^0 \vee F^1 \vee F^2 \vee \ldots$$

we will have yF^*x if and only if for some n, $y = F^n:x$. In general, there may be many such n, so F^* may not be left univalent. If F^* is to be a function, it is necessary to pick a <u>termination condition</u> T (a class) that is only true for one of $F^0:x$, $F^1:x$, $F^2:x$, ... Then $T \nless F^*$ is just the function sought; it is roughly equivalent to

<u>while</u> ~T <u>do</u> F

Analogously, $T \nless F^+$ is roughly equivalent to repeat F until T

13. Arrays

13.1 definition and basic operations

An array is just a function from a contiguous subset of the integers to some set of values. If A is an array and i Rm A then A:i is the i-th element of A. Similarly, if I $\subseteq$ $\overrightarrow{\text{Rm}}$:A is a set of index values then A!:I is the corresponding set of array values and A}I is the subarray of A selected by those indices.

It is easy to define multi-dimensional arrays: they are just arrays whose elements are selected by sequences of integers, e.g. M:(i,j). If M is a two-dimensional array, then M(i,) is the i-th row of M and M(,j) is the j-th column of M. Also, if I is a set of row indices and J is a set of column indices then M}(I$\times$J) is the submatrix of M selected by these sets. It is easy to see that M' is the transpose of M, since

M':(i,j) = M:(':(i,j)) = M:(j,i)

More generally, if P is a permutation function (i.e. a bijection from an index set into itself) then AP is the result of permuting A by P.

Suppose x is an element of the array A (i.e., for some i, x=A:i). Then $\overleftarrow{A}$:x is the set of all indices for which x=A:i. Therefore we can find the index of the first occurence of x in A (i.e. APL's iota operator) by min$\overleftarrow{A}$:x. In general, if P is some property (i.e. class), then A^{-1}!:P is the set of indices of all elements of A that satisfy P. A sorted reflexive sequence of these indices is just $\leq$ ӿ (A^{-1}!:P)

13.2 relation to sequences

It is easy to convert arrays to sequences and vice versa. Suppose all the elements of A are distinct, then A^{-1} is a function that returns the index of an element of A. We want to define a sequence S such that xSy if and only if x preceeds y in A, i.e. the index of x is one less than the index of y.

```
xSy  <->  (A^-1:x)  =  (A^-1:y)-1
     <->  (A^-1:x) (-1) (A^-1:y)
     <->  x[A|(-1)|A^-1]y
```

Hence, S = A(-1)A^{-1}.

Next, we will consider the opposite process: converting a sequence to an array. Suppose we have a sequence:

$$\begin{tikzcd} S = a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \end{tikzcd}$$

We wish to convert this to an array:

$$\begin{tikzcd}[row sep=tiny] & a_0 \arrow[r] & 0 \\ A = & a_1 \arrow[r] & 1 \\ & a_2 \arrow[r] & 2 \\ & a_3 \arrow[r] & 3 \end{tikzcd}$$

Thus, for each element a_i in the sequence, we must find its index i in the resulting array. If we can define a relation R such that $R{:}a_i{=}i$ then R^{-1} will be the array we seek. Now $R{:}a_i$ is just the number of predecessors of a_i in S. That is, a_0 has no predecessors, so $R{:}a_0 = 0$; a_2 has two predecessors, so $R{:}a_2 = 2$, and so on. Since S defined an immediate predecessor relation, S^+ defines an ancestral predecessor relation:

$$\begin{tikzcd} S^+ = a_0 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left=50] & a_1 \arrow[r] \arrow[rr, bend left] & a_2 \arrow[r] & a_3 \end{tikzcd}$$

The set of predecessors of any element a is then $\overrightarrow{S^+}{:}a$, e.g.

$$\overrightarrow{S^+}{:}a_2 = \{a_0, a_1\}$$

The size of this class is then the desired index:

$$\#{:}(\overrightarrow{S^+}{:}a_2) = 2$$

Hence, $R{:}a = \#{:}(\overrightarrow{S^+}{:}a)$, so $R = \#\overrightarrow{S^+}$. Now, we know that A is R^{-1}, so we can define the function sa0 which converts a sequence into a 0-origin array:

$$sa0{:}S = (\#\overrightarrow{S^+})^{-1}$$

To produce a 1-origin array, the only alteration is:

$$sa{:}S = (\#\overrightarrow{S^*})^{-1}$$

13.3 <u>other array operations</u>

Next we will consider the concatenation of arrays. If A is an array such that $A{:}i = a_i$, then we can write A:

$$A = (a_1,1) \vee (a_2,2) \vee \cdots \vee (a_m,m)$$

where m is the length of the array. Similarly, suppose that B is an n element array, then the concatenation of these arrays is

$$A \text{ cat } B = (a_1, 1) V \cdots V (a_m, m) V (b_1, m+1) V \cdots V (b_n, m+n)$$

We can see that A cat B = AVB' where B' results from B by shift its indices by m:

$$B' = (b_1, m+1) \; V \cdots V \; (b_n, m+n)$$

How do we compute B'? Observe:

$$xB'i \;\leftrightarrow\; xB(i-m) \;\leftrightarrow\; xB[(-m):i] \;\leftrightarrow\; xB(-m)i$$

Hence, B' = B(-m) and A cat B = A V B(-m), where m is the length of A. The length of A is just #rim:A, so

$$A \text{ cat } B = A \; V \; B(-\#\text{rim}:A)$$

We will finish our discussion of arrays by investigating the generation of sorted arrays. Let S be a set of integers to be sorted, then $\leq \ast S$ is a structure which relates lesser elements to greater elements. Now if x is any element of the set, $\overrightarrow{(\leq \ast S)}:x$ is the set of all elements less than x. Thus $[\#\overrightarrow{(\leq \ast S)}]:x$ is the number of elements of S less than or equal to x. This is just the index of x in the sorted array we seek. Hence if A is the sorted array, xAi if and only if $i[\#\overrightarrow{(\leq \ast S)}]x$, so $A = [\#\overrightarrow{(\leq \ast S)}]^{-1}$. Of course this can be generalized to any ordering relation.

14. Scanning Structures

14.1 basic concepts

In this section we will discuss several methods for scanning structures, that is, for applying a function to each element of a structure and accumulating the results. Since no one method has yet been selected, this section should be taken as a report of work in progress.

A general paradigm for processing a structure, such as a file, is the following:

1. Perform some initialization.

2. Read the next (or first) element of the file.

3. Take this value and the results of processing the previous values.

4. Process these to yield new cumulative values and continue from step (2).

5. When the end of the file is reached, return the accumulated result of processing all of its elements.

A simple form of this appears in APL's reduction operation:

$$+/V \quad = \quad V_1+(\ldots(V_{n-1}+V_n)\ldots)$$

A more general form is Backus' insert:

$$/f{:}\langle x_1,\ldots,x_n\rangle \quad = \quad f{:}\langle x_1,\ldots\ f{:}\langle x_{n-1},x_n\rangle\ldots\rangle$$

Our first example of scanning structures will be to express this operation in the relational calculus.

14.2 <u>reduction of arrays</u>

We are given an n element array A and wish to compute:

$$t \quad = \quad A{:}n + A{:}(n-1) + \ldots + A{:}2 + A{:}1$$

where we have assumed that the right members of A are 1..n. We saw in the section on ancestrals that $T\langle F^*$ will iterate the application of F with T used as the termination condition. Consider how the analogous loop would be written in Pascal:

```
S := 0;  i := 0;
while i≠n+1 do
  begin  S := S+A[i];  i := i+1  end
```

On each iteration two functions are performed: S is incremented by A[i] and i is incremented by 1. Let's represent the state of the computation by a pair (s,i), where s is the cumulative sum so far and i is the index of the next element to process. We will use F to represent one processing step, so that, if (s',i') is the new state, we can solve for F as follows:

$$\begin{aligned} F{:}\begin{pmatrix} s \\ i \end{pmatrix} &= \begin{pmatrix} s' \\ i' \end{pmatrix} \\ &= \begin{pmatrix} s+A{:}i \\ i+1 \end{pmatrix} \\ &= \begin{pmatrix} (+)\frac{I}{A}{:}\begin{pmatrix} s \\ i \end{pmatrix} \\ (+1){:}\ i \end{pmatrix} \\ &= \begin{pmatrix} (+)\frac{I}{A}{:}\ (s,i) \\ (+1)\omega{:}\ (s,i) \end{pmatrix} \\ &= \left.\frac{(+)\frac{I}{A}}{(+1)\omega}\right|{:}\ \begin{pmatrix} s \\ i \end{pmatrix} \end{aligned}$$

$$\text{Hence, } F \quad = \quad \left.\frac{(+)\frac{I}{A}}{(+1)\omega}\right|.$$

It remains to determine the termination condition, T. If x is a state, i.e., a pair (s,i), then $x \in T$ when i=n+1. Hence, $x \in T$ when $\omega : x = n+1$, so

$$
\begin{aligned}
x \in T &\leftrightarrow \omega : x = n+1 \\
&\leftrightarrow (n+1)\ \omega\ x \\
&\leftrightarrow x \in \overleftarrow{\omega} : (n+1)
\end{aligned}
$$

Hence, $T = \overleftarrow{\omega} : (n+1)$. The final state, x_f, containing the sum is $T \nless F^* : x_i$, where $x_i = (0,1)$ is the initial state:

$$x_f = (T \nless F^*) : (0,1)$$

Now, the total t is just $\alpha : x_f$, so

$$t = \alpha (T \nless F^*) : (0,1)$$

We can generalize this to any function f with initial value i:

$$t = \alpha (T \nless F^*) : (i,1)$$

$$\text{where } F = \left. \frac{f \frac{I}{A}}{(+1)\omega} \right|$$

This result can be improved by directly extracting the result from the final state. That is, we want to define a filter $\varnothing$ such that $t = \varnothing F^* : (i,1)$. Hence we want $t \varnothing x_f$, so

$$t \varnothing x_f \leftrightarrow t\ \varnothing\ (t,n+1)$$

Now, note that $[\,,n+1] : t = (t,n+1)$, so

$$(t,n+1)\ [\,,n+1]\ t$$

by the definition of ':'. Therefore $\varnothing = [\,,n+1]^{-1}$ and we have the simplified formula

$$t = (\,,n+1)^{-1} F^* : (i,1)$$

14.3 <u>reduction of sequences</u>

Next we will consider the scanning of sequences. Suppose S is a sequence:

$$S = \langle s_1, s_2, \ldots, s_n, EOF \rangle$$

where EOF is an "end marker"; it can be any value. Now, we wish to find the result

$$i\ f\ s_1\ f\ s_2\ f\ \ldots\ f\ s_n$$

that is

$$f:(\ f:(\ldots\ f:(\ i,\ s_1\ )\ \ldots),\ s_n)$$

for some function f and starting value i. The state can be represented by a pair (t,s), where t is the result so far computed and s is the rest of the sequence to be processed. Hence, (t',s') = F:(t,s) where t' = f:(t,α:s) and s' = $\underline{\Omega}$:s. Therefore,

$$\begin{pmatrix} t' \\ s' \end{pmatrix} = \begin{pmatrix} f:(t,\alpha:s) \\ \underline{\Omega}:s \end{pmatrix} = \begin{pmatrix} f\frac{I}{\alpha}: & (t,s) \\ \underline{\Omega}\omega: & (t,s) \end{pmatrix} = \left.\begin{array}{c} f\frac{I}{\alpha} \\ \hline \underline{\Omega}\omega \end{array}\right| : \begin{pmatrix} t \\ s \end{pmatrix}$$

Hence,

$$F = \left.\begin{array}{c} f\frac{I}{\alpha} \\ \hline \underline{\Omega}\omega \end{array}\right|$$

What is a terminal state? Notice that $\underline{\Omega}$:<s_n,EOF> = $\ominus$, so a terminal state will have the form (r,$\ominus$). Hence,

$$r = (,\ominus)^{-1}F^{*}: (i,S)$$

To put this in a more useful form, we will define a function f@i such that r = (f@i):S. This is simply

$$f@i = (,\ominus)^{-1} \left.\begin{array}{c} f\frac{I}{\alpha} \\ \hline \underline{\Omega}\omega \end{array}\right|^{*} (i,)$$

Then, the sum of the elements of a sequence S is just (+)@0:S.

14.4 <u>scanning general structures</u>

It is often useful to scan a structure while performing some processing at each node. When the data structure is a sequence this amounts to APL's reduce operator and Backus' insert operator. We will define a scanning operation that works on a more general class of structures. This operator can be understood intuitively as follows: The state of the scanning process is represented by a set of "read heads" each of which is "positioned over" a node and holds state information accumulated from the nodes it has already visited. A node can be processed when a read head has moved to that node over each edge which leads into the node. When this occurs a processing function is applied to the node (as first parameter) and the union of the state information of each of the read heads (as second parameter). The result of this processing step becomes the state information associated with a new set of read heads which are advanced along each edge leading out from the node. The processing of the structure is completed when all read heads have arrived at terminal nodes (hence this scanning operation is not defined for cyclic structures). Scanning a structure is started by positioning a read head with initial state information over each initial node.

The scanning operation is symbolized by f]i, where f is the processing function and i is the initial state for the read heads. For instance, if V is a vector, (+)]0:V will scan the elements of V using (+) (i.e. APL +/V or Backus' (/+):V). For a more interesting example, suppose T is an attributed parse tree, E is a function that evaluates attributes and B is the initial set of attribute bindings. Then E[B:T propogates the values of inherited attributes down to the leaves of the tree. Conversely, E[B:(T^{-1}) propogates the values of synthesized attributes back to the root. Hence, repeated applications of E[B and (E[B)' will evaluate all of the attributes. Of course, this program will work just as well if T is a forest of parse trees. The [operator is still undergoing evaluation as it is one of several possible structure-directed scanning operations.

15. Examples

In this section we will give several examples of relational programs.

15.1 payroll

Suppose we have a file Φ of employee records, where $r = \Phi:n$ is the record for the employee with the employee number n. We will suppose that employee records are functions defined so that:

r:N	=	employee name
r:H	=	hours worked so far this week
r:R	=	pay rate

We are given an update file U such that U:n is the number of hours worked by employee n today. We wish to generate a new payroll file Φ'.

SOLUTION: Let $r = \Phi:n$ and $r' = \Phi':n$ be the old and new employee records. It is clear that r' is the same as r except for its H field. In order to modify part of a relation, we will use the Md function defined by:

$$\mathrm{Md}:(S,R) \;=\; R \vee S \succ (-\overrightarrow{Rm}:R)$$

Then, if h' represents the new value of the H field, the new employee record is

$$r' \;=\; \mathrm{Md}:(r,\ (h',H))$$

where h' is just the cumulative hours worked:

$$h' \;=\; (\Phi:n):H + U:n$$

Therefore, by the definition of Φ':

$$\Phi':n \;=\; r' \;=\; \mathrm{Md}:(\ \Phi:n,\ (h',H))$$

To find Φ' we must factor out the employee number n. To do this, note that $(\Phi:n):H = (:H):(\Phi:n) = (:H)\Phi:n$. That is, $(:H)\Phi$ is a slice of the payroll file: the hours worked for each employee. Therefore,

$$h' = (\Phi:n):H + U:n = (:H)\Phi:n + U:n$$
$$= (+)\left.\frac{(:H)\Phi}{U}\right|:n$$

Now, define the updating function u by

$$u:n = \left((+)\left.\frac{(:H)\Phi}{U}\right|:n,\ H \right)$$
$$= (,H)\,(+)\left.\frac{(:H)\Phi}{U}\right|:n$$

Then, $\Phi':n = Md:(\Phi:n,u:n) = Md\left.\frac{\Phi}{u}\right|:n$ Therfore, the solution to our problem, the new payroll file, is

$$\Phi' = Md\left.\frac{\Phi}{u}\right|$$
$$\text{where } u = (,H)\,(+)\left.\frac{(:H)\Phi}{U}\right|$$

15.2 <u>check issueing</u>

Suppose we wish to take the payroll file from the previous example and generate checks for the employees. We will assume that a function C is available such that C:(nm,p) returns a check in the amount p made out to the name nm.

SOLUTION: We will ignore overtime computations. Hence, if n is an employee number then $\Phi:n:N$ is his name and

$$p:n = \Phi:n:H * \Phi:n:R$$

is his pay. Hence, his check c:n is $c:n = C:(nm,p:n) = C:\begin{pmatrix} nm \\ p:n \end{pmatrix}$

$$= C:\begin{pmatrix} (:N)\Phi:n \\ p:n \end{pmatrix} = C\left.\frac{(:N)\Phi}{p}\right|:n$$

Combining these we have the file F mapping employee numbers into checks:

$$F = C\left.\frac{(:N)\Phi}{{}^{*}_{-}\left.\frac{:H}{:R}\right|\Phi}\right|$$

from which we can factor out the old payroll file:

$$F = C\left.\frac{:N}{{}^{*}_{-}\left.\frac{:H}{:R}\right|}\right|\Phi$$

If we just want a set of checks, this is $\overrightarrow{Lm}:F$.

16. <u>Implementation Notes</u>

The primary goal of our investigation has been to determine if relational programming is significantly better than conventional methods. It would be premature to devote much effort to implementation studies before it is even determined if relational programming is an effective programming methodology. However, a brief discussion of implementation possibilities is probably not out of line.

The most obvious representation of a relation is the _extensional_ representation, in which all the elements of a relation or class are explicitly represented in memory. There are many kinds of extensional representations, such as hash tables, binary trees and simple sorted tables. Of course, performance can be improved through the use of associative memories and active memories (in which each memory cell has a limited processing capability).

Some relations and classes will be so large that it is uneconomical to represent them explicitly in memory. In these cases an _intensional_ representation [11] should be used. Here a class or relation is represented by a formula or expression for computing that relation or class. Operations on the class or relation are implemented as _formal_ operations on the expression. This is feasible because of the simple algebraic properties satisfied by relations. It can be seen that an intensional representation is really just a variant of a _lazy evaluation_ mechanism [9, 10]. Sometimes an intensional representation is necessary; for instance, relations of infinite cardinality, such as the numerical operators and relations, require an intensional representation.

Although the programmer could be allowed to choose between extensional and intensional representations for his relations, this is not necessary. It is probably feasible, and certainly higher level, to have the system choose representations on the basis of cardinality estimates of the classes and relations involved. The algebra of relations is regular enough that many of these decisions can be made at compile time. Any that can't can be deferred to run-time when exact cardinality information is available. See [14] for related techniques.

17. Conclusions

Of course, we are not the first to propose introducing aspects of a relational calculus into programming. Codd [4] has used a relational calculus as the basis for data base systems. Although he defines several operations on relations (_viz_., premutation, join, tie, composition, and restriction), this small set of operations is insufficient for general purpose programming. These remarks also apply to Childs' reconstituted definition of relations [3], which are also oriented towards data bases. Feldman and Rovner [6] augmented Algol with several relational operators for associative access to a data base. Their operations, which are our plural description and image, are quite limited, being based on a traditional von Neumann language.

One general purpose language that does make extensive use of sets and relations is SETL [7], which provides most of the familiar operations on sets (e.g., union, intersection, difference, powerset, image). SETL differs from relational programming in three significant respects: (1) it can only handle finite sets, (2) many operations must still be performed in a word-at-a-time fashion using the set former, and (3) it resorts to conventional control structures.

Finally, we must mention "logic programming" systems, such as PROLOG [15, 8], which use predicate logic to describe computational processes. These systems also differ from relational programming in several significant respects: (1) they have a word-at-a-time programming style due to the use of variables representing individuals in the clauses of the program, and (2) they are implemented using a resolution theorem prover, whereas a more conventional procedural implementation suffices for relational programming. Essentially the same remarks apply to Popplestone's relational programming [13], which is like logic programming except that it uses "forward inference" rather than "backward inference".

In summary, no other programming style that we are aware of combines the universal use of relations with a rich set of operations on those relations that can be implemented in a deterministic, procedural way. It is hoped that the preceeding discussion has made plausible some of the advantages claimed for relational programming in the Introduction. Considerable work remains to be done in evaluating the effectiveness of a relational calculus as a programming tool. For instance, the optimum set of combinators and relational operators must be selected. Another non-trivial problem is the selection of a good notation for the relational calculus. More from convenience than conviction we have used the notation of [16] and [2]. Making relational programming an effective tool will require designing a notation that combines readability with the manipulative advantages of a two-dimensional algebraic notation. This is all preliminary to any serious considerations of software or hardware implementation techniques.

18. References

[1] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, CACM 21, 8 (August 1978), 613-641.

[2] Carnap, R. Introduction to Symbolic Logic and its Applications, Dover, 1958.

[3] Childs, D.L. Feasibility of a set-theoretic data structure based on a reconstituted definition of relation. IFIP 68 Proceedings, 420-430, North-Holland, 1969.

[4] Codd, E.F. A relational model for large shared data banks, CACM 13, 6 (June 1970), 377-387.

[5] Curry, H.B., Feys, R. and Craig, W. Combinatory Logic, I, North-Holland, Amsterdam, 1958.

[6] Feldman, J.A. and Rovner, P.D. An Algol-based associative language, CACM 12, 8 (August 1969), 439-449.

[7] Kennedy, K. and Schwartz, J. An introduction to the set theoretical language SETL, J. Comptr. and Math. with Applications 1 (1975), 97-119.

[8] Kowalski, R. Algorithm = logic + control, CACM 22, 7 (July 1979), 424-436.

[9] Henderson, P. Functional Programming Application and Implementation, Prentice-Hall, 1980, 223-231.

[10] Henderson, P. and Morris, J.H., Jr. A lazy evaluator, Record 3rd ACM Symp. on Principles of Programming Languages, 1976, 95-103.

[11] MacLennan, B.J. Fen - an axiomatic basis for program semantics, CACM 16, 8 (August 1973), 468-474.

[12] MacLennan, B.J. Introduction to Relational Programming, Computer Science Department Technical Report NPS52-81-008, Naval Postgraduate School, 1981.

[13] Popplestone, R.J. Relational programming, in Hayes, J.E. et al. (eds.), Machine Intelligence 9, Halsted Press, 1979, 3-26.

[14] Schwartz, J. Automatic data structure choice in a language of very high level, CACM 18, 12 (December 1975), 722-728.

[15] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language, JACM 23, 4 (October 1976), 733-742.

[16] Whitehead, A.N. and Russell, B. Principia Mathematica to *56, Cambridge, 1970.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Gordon H. Bradley Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	20

DATE
ILME